A PROOF-OF-CONCEPT FOR SOFTWARE-ONLY ATTESTATION ON REAL-TIME SYSTEMS USING VON NEUMANN ARCHITECTURE AND DYNAMIC MEMORY ALLOCATION

THESIS

Travis S. Potthoff, Captain, USAF

AFIT-ENG-MS-17-M-062

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-17-M-062

A PROOF-OF-CONCEPT FOR SOFTWARE-ONLY ATTESTATION ON REAL-TIME SYSTEMS USING VON NEUMANN ARCHITECTURE AND DYNAMIC MEMORY ALLOCATION

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Cyber Operations

Travis S. Potthoff, BS

Captain, USAF

March 2017

AFIT-ENG-MS-17-M-062

A PROOF-OF-CONCEPT FOR SOFTWARE-ONLY ATTESTATION ON REAL-TIME SYSTEMS USING VON NEUMANN ARCHITECTURE AND DYNAMIC MEMORY ALLOCATION

Travis S. Potthoff, BS

Captain, USAF

Committee Membership:

Scott R. Graham, Ph.D.
Chair

Barry E. Mullins, Ph.D., P.E.
Member

Maj Timothy J. Carbino, Ph.D.
Member

AFIT-ENG-MS-17-M-062

Abstract

To attest is to affirm to be correct, true, or genuine. Applied to software or executable code, attestation is the ability to affirm that the code actually being executed is the code that is expected, unmodified in any way and may be performed in either hardware or software. Current research into software-based attestation has explored the problem of static attestation, or verifying the software that the system loads at boot-time. For many systems, knowing that the system's initial state is valid is insufficient – verification that the system is still in a good state is needed later and without bringing the system offline or interrupting critical processes. This thesis introduces a proof-of-concept method for performing attestation on real-time systems, named Dynamic Attestation of Run-Time Systems (DARTS). DARTS was designed to be sufficiently customizable in order to enable attestation without interfering with system operations. DARTS also has the ability to perform attestation on systems built upon Von Neumann architecture using dynamic memory allocation. A key contribution of this work is that the entire attestation process is performed wholly in software on a real-time system without impacting the operation of potentially critical processes.

Acknowledgments

I would like to thank my dear wife for putting up with my incessant typing and permanent residence at my desk while I worked on this thesis. That, and taking care of our dragon-baby while I was hard at work.

I would also like to thank my advisor for his guidance during this process, both on the thesis as well as in military matters. And to my classmate and friend, Capt R. C., for his help while dealing with personal issues that impacted our group project.

And to you, little daughter, thank you for bearing with me and not turning off my computer as I work to keep us housed and fed.

Travis S. Potthoff

Table of Contents

Page

List of Figures

Page

List of Tables

Page

A PROOF-OF-CONCEPT FOR SOFTWARE-ONLY ATTESTATION ON REAL-TIME SYSTEMS USING VON NEUMANN ARCHITECTURE AND DYNAMIC MEMORY ALLOCATION

1. Introduction

1.1 Background and Motivation

In a computer age, where computers outnumber people and are used to coordinate and control various aspects of their lives, the ability to trust that a particular computer system has not been tampered with becomes increasingly essential. Whether the system is medical equipment monitoring a patient on life support or the autopilot systems on a passenger airliner, malicious logic can quickly turn devices from asset to liability with catastrophic consequences.

To attest is to affirm to be correct, true, or genuine. Applied to software or executable code, attestation is the ability to affirm, or verify, that the software running on a system is what the user or administrator expects; nothing more and nothing less. Research into attestation has proven that methods performed solely in software can be used to ensure that the system is operating as expected, but such methods universally require that the system's operation be interrupted, even if just temporarily, in order to perform the necessary validation (Seshadri et al., 2005).

SoftWare-based ATTestation (SWATT) is one of the foundational works in software-based attestation, effectively proving that such attestation is possible [2]. Unfortunately, for the algorithm to detect alterations to the code, the system must surrender control to SWATT for the duration of the attestation because any interruptions

in the execution of software-based attestation can grant malicious logic the ability to circumvent the attestation scheme and effectively hide itself. In the case of SWATT in particular, the entire memory of the system that is to be verified is incorporated into the check, which means the downtime required for attestation would be noticeable and possibly disruptive.

In real-time systems, such as critical infrastructure and vehicle control systems, the problem of attesting software becomes more complicated. Bringing such systems offline, even for an instant, could have catastrophic consequences. Dynamic Attestation of Real-Time Systems (DARTS) [3] is a proof-of-concept attestation method that seeks to explore the possibility of using software-based attestation methods at run-time on real-time systems without impacting the operation of the systems. Using this, it would be possible to verify that critical systems are operating as intended without the need for a service interruption.

1.2 Problem Statement

As was demonstrated by the Aurora Generator Test [4] and *Comprehensive Experimental Analyses of Automotive Attack Services* [5], real-time systems such as critical infrastructure and vehicles with onboard computers are vulnerable to cyber incursions, with potentially catastrophic results. Current methods of software attestation either require additional custom hardware or interruption to the system's operation in order to execute, which could possibly cause system instability. It is for this reason that Dynamic Attestation of Real-Time Systems (DARTS) was developed – a software

attestation scheme conducted purely in software that can run without interruption to the critical processes.

1.3 Research Objectives /Hypotheses

Using DARTS as a representative algorithm for software attestation, this research aims to confirm the hypothesis that software attestation is possible on real-time systems as well as systems built on a Von Neumann architecture with dynamic memory allocation. In order to enable DARTS to emulate the behavior of existing attestation methods using available hardware, it was also necessary to account for the differences between Harvard and Von Neumann architecture as well as static versus dynamic memory allocation with respect to software attestation. As these differences were evaluated, investigation into feasibility of performing software attestation in such environments was performed, leading to the hypothesis that software attestation is feasible on systems utilizing dynamic memory allocation and a Von Neumann architecture.

1.4 Research Focus

This research focuses on determining the feasibility of interleaving software-only attestation with real-time processes. This includes a verification of timing deconfliction between the process and attestation, and assessing the impact of a reduction in the attestation scope on the security efficacy of attestation. In order to implement software attestation on the available systems, DARTS has been coded to account for dynamic memory allocation as well as a Von Neumann hardware architecture, which further expand the types of systems software attestation is capable of being performed on.

1.5 Investigative Questions

This research aims to answer several key questions. The first of which is whether or not performing software-only attestation on a real-time system is even possible. In the event that attestation is possible but must be reduced to remain non-disruptive, how many iterations of attestation must be performed in order to guarantee that alterations have been found? Finally, what sort of special considerations or alterations must be made in order to accommodate for a Von Neumann architecture with dynamic memory allocation?

1.6 Methodology

This proof-of-concept utilizes two PC-in-a-box systems configured with a real-time Linux operating system as the testbed devices. This system configuration allows for nanosecond level scheduling and should effectively represent scheduling-based real-time systems. Then, using a program designed to represent a real-time process and DARTS, the attestation program, to perform tests and demonstrate the feasibility of real-time attestation and assess the implications of altering the attestation space in order to create flexibility in the attestation scheme.

1.7 Implications

Successful demonstration that attestation can be dynamically performed on real-time systems is the first step towards enabling system owners to incorporate a dynamic attestation scheme to guarantee confidence in the system during all phases of its execution. Due to the increased reliance on computers to handle critical infrastructure, determining if such systems have been altered by an adversary could mean the difference, in some cases, between life and death.

1.8 Preview

This thesis is structured as follows: Chapter 2 introduces related works that have established the software-based attestation paradigm, as well as alternate methods that require more significant modifications to the system. An overview of the methodology used to conduct the experiments is presented in Chapter 3, including assumptions, system specifics, and an overview of DARTS functionality and limitations. Chapter 4 contains an analysis of the results of the experiments, as well as potential implications. Finally, Chapter 5 presents a summarized conclusion of the results and highlights of areas of interest for future work.

2. Literature Review

2.1 Chapter Overview

The purpose of this chapter is to introduce foundational works regarding software attestation and discuss their relevance to this research.

2.2 Relevant Research

Prior to coding and experimentation, it was necessary to research previous forays into the realm of software attestation. Software attestation research can generally be divided into two sub-categories: software-only attestation and hardware-assisted attestation. As the focus of this thesis is to research software-only attestation, proposed methods that include specialized or additional hardware proved to be of little use because every attestation method where specialized hardware is used, the hardware is required to provide some form of localized security – to prevent malware from having access to either the attestation method or the nonce (number used once). When using software-only methods, such an approach is inapplicable. In fact, one of the stated assumptions for software-only attestation is that the adversary has full access to the entire process.

2.2.1 SWATT, Extensions, and Derivative Works

The most influential works relating to software-only attestation stem from S*oft*W*are-based* ATT*estation (SWATT)* [2]. SWATT (and all derivative methods mentioned) is a client-server architecture where the verification system sends a nonce to the remote client that is to be verified. The client then uses the nonce as a random seed to create a deterministic hash of the entire memory of the client within a pre-determined amount of time. If either the hash is incorrect or the response is too late, the system is

flagged as compromised. The benefit of this approach is that all of the client's memory is evaluated, including any non-used space, which is filled with pseudo-random noise to prevent malware from concealing itself within the zeroes. The obvious downside is that the entire system has to surrender control for the duration of the attestation, making a full system memory attestation costly.

There have been papers such as *On the Difficulty of Software-Based Attestation of Embedded Devices* [6], in which the authors describe certain flaws in each of the algorithms discussed (primarily SWATT and ICE). The authors used hardware that did not match the original designer's systems in order to add the element of portability to the mix, which at least for SWATT did pose a possible area for concern. Additionally, the original authors of *SWATT* published a refutation [7] that not only addresses the concerns brought up in [6] but produced references pointing to previous works either demonstrating that the concerns had been addressed or admitting to them directly. The primary benefit of these papers was to illuminate the intricacies of software-based attestation with a back-and-forth discussion that introduced and addressed issues with the attestation methodology.

F*orgery-resistant* I*ntrusion detection*, R*ecovery, and* E*stablishments of keys (FIRE) and* I*ndisputable* C*ode* E*xecution (ICE)* [8] is the next step in software attestation stemming from SWATT. Rather than evaluating the entire memory contents of the system, ICE first attests its own code resident on the remote system, which then attests the program (or programs) desired. FIRE, aside from appearing in the paper's title and briefly in methodology, exists more as a supportive element to ICE. This reduces the scope of attested memory from that of the entire system to only the selected program(s)

(or a subset of a single program). ICE can greatly reduce the amount of attestation performed, making it far more suitable for larger systems and possibly even extended into the real-time domain.

Pioneer [1], *Secure Code Update by Attestation* (SCUBA) [9], and *SAKE: Software Attestation for Key Establishment in Sensor Networks* [10], extend the idea of using ICE for software attestation. Pioneer uses the same general format as ICE, with the key difference being that Pioneer utilizes SHA-1 rather than alternating ADDs and XORs. To compensate for SHA-1's collision resistance having been broken, the authors use a provided nonce to reduce the feasibility of pre-computation, as malware would have to pre-compute the hash independently for every nonce available. SCUBA, on the other hand, demonstrates the use of software-based attestation as a method for verifying that updates had been performed properly on an untrusted system via an implementation of ICE. In a similar manner to SCUBA, SAKE utilizes ICE in order to implement a secure key exchange protocol despite the remote system being in an uncertain state.

The work presented in [11] predominantly relies upon the SWATT evaluation and proof of concept but with altered code in order to show that it will function appropriately on vehicle systems. Malicious alterations to the verification code would incur an estimated 13% overhead that would be detectable by the verification system. The authors do not perform an in depth discussion of the actual results, leaving much to speculation. The primary benefit of this paper is that it displays the transferability of SWATT between platforms, demonstrating that the original proof of concept is not bound by a narrow scope of systems.

2.2.2 Alternative Software-based Methods

A method specifically targeting inclusion of the stack into attestation is proposed in [12], wherein the system replaces the entire contents of memory with a deterministic pseudorandom number, which would greatly impact any ongoing operations on the system. This allows the entire contents of the system's memory to be attested, but completely precludes the possibility that this approach can be used on a real-time system as it effectively resets the system to startup configuration resulting in a loss of any data in use or that has been collected. This technique is also used in [13], where the authors openly state that "Firmware attestation might not be suited for devices with harsh time constraints" due to the likelihood of it being time consuming for the device. An idea that this paper rejected was the concept of searching through memory for data that resembles executable code – a concept that could prove quite useful in detecting malware that hides itself on the stack. Unfortunately, no details were presented regarding this line of thought.

Some groups have explored the possibility of performing system state verification at run-time. [14] and [15] both focus on run-time verification of the system state. The primary benefit to these two approaches is that they perform some degree of hardware verification in addition to software, but at a cost. Both methods' verification process is focused on checking input / output validity on the target system – that is checking to see if the expected output is produced for a given input. While this can be used to indicate hardware or software changes that affect system operation, they are not able to discover changes that seek to remain hidden. Malware waiting for a trigger condition, for example, would not be caught by either method until the malware had already executed.

Additionally intelligent malware could return the expected output to the verifier while still carrying on with its nefarious purposes.

2.2.3 Problems with Software Attestation Methods

In *A Generic Attack on Checksumming-Based Software Tamper Resistance* [16], the authors implemented proof-of-concept versions of attacks on various operating systems and architectures and then tested to see if their modifications to the system were detected by a "representative checksumming tamper resistance algorithm". Unfortunately, the authors never specified the capabilities that were built into the representative checksumming algorithm, greatly limiting the transferability of the results. The attack methods involve duplicate copies of the program code (one modified, one not), meaning that at least enough empty space is required to store the program code. While this may be viable in larger systems, it is problematic in small, embedded devices.

Another paper that focused on the flaws in software attestation is *Side Effects Are Not Sufficient to Authenticate Software* [17], in which the authors assess the ability of different attestation methods. Without access to the source code, as they repeatedly state, the accuracy of their results is questionable, at least with regards to Genuinity [18], the primary focus of their research. In their assessment of SWATT, however, they list a few concerns about its ability to guarantee memory integrity but concur with the conclusion that SWATT, if used as intended, would result in malware detection with a very high degree of probability.

2.2.4 Hardware Assisted Methods

In *A Minimalist Approach to Remote Attestation* [19], the authors propose a generic approach to remote attestation, which differs from typical software attestation in

that it is meant to be performed over a network as opposed to being directly connected. This approach is not timing-based. Rather, it relies on the creation of a secure attest function and statistics for the provided nonce to be integrated into the function, computed, and then returned with negligible chances of collision or guessing. The method proposed is entirely time agnostic; therefore, it can be performed over a network without fear of the other network devices interfering with the attestation process. In the end, this paper relies upon hardware support (exclusive access to the nonce) in order for the attestation to be effective.

Alternatively, instead of simply relying on the system to verify itself while using possibly corrupted programming, solutions like *Dynamic Integrity Measurement and Attestation* [20] and *New Results for Timing-Based Attestation* [21] propose that a trusted platform module (TPM) be added to the system. The TPM is essentially a co-processor with read-only memory containing the code that is used to attest the remainder of the system. This presents the obvious benefit in that it can be guaranteed that the component performing the attestation has not been co-opted to function differently, assuming no changes have been made to the system's hardware. Unfortunately, this requires the installation of additional hardware into the system and increases the complexity of performing system upgrades while maintaining an attestable state.

2.2.5 Attacks Against Attestation

In *TOCTOU, Traps, and Trusted Computing* [22] the authors wrote code that demonstrated the *T*ime *o*f *C*heck, *T*ime *o*f *U*se (TOCTOU) style attack by changing functions at run-time, rather than at boot-up, and then reverting the changes once the malicious tasks were complete. Malware acting in this manner effectively defeats

attestation methods that only check either at boot-up or during a system interruption. It is for this reason DARTS was created - the purpose of run-time attestation is to eliminate the disparity between the time of check and time of use referenced in the paper.

2.3 Summary

Research in the realm of software-based attestation has verified that attestation, in various forms, is possible and ranges from small embedded devices to larger systems. However, there has been little research into the possibility of implementing attestation into a running system without disruption to the system's operations.

3. Methodology

3.1 Chapter Overview

The purpose of this chapter is to present the process followed to conduct the experiments. This starts with a description of the DARTS program, then outlines assumptions and limitations for the program as well as software attestation in general, states the criteria that will be used to measure success, and then introduces the system configuration used.

3.2 Dynamic Attestation of Real-Time Systems (DARTS)

In order to explore the possibility of performing dynamic (i.e., run-time) attestation on real-time systems, the ICE algorithm developed by Seshadri, et al., was extended to create DARTS. ICE was chosen in order to capitalize on the ability to focus attestation on a single process without the need for additional hardware. The process by which DARTS performs attestation is displayed in Figure 1. Attestation begins with the verification server transmitting a randomly generated nonce to the client to be attested. At the same time, the verification server starts a timer. When the nonce is received by the client, DARTS utilizes system calls to determine the memory addresses associated with the desired program. DARTS then uses the nonce to seed a pseudo-random number generator to determine which memory addresses will be included in the hash and, using the nonce as the initial value for the hash, hashes the values of the selected addresses sequentially. DARTS then uses the current hash value to re-seed the number generator to select the Target System Process memory addresses and incorporates the selected values into the hash. Once all desired addresses have been included in the hash, DARTS

transmits the hash to the verification server which ensures that the hash is the expected value and it was received within the proper amount of time. The hash used in DARTS is not intended or required to have strong cryptographic security properties; rather its primary function is to increase complexity required for an adversary to pre-compute the response that will be sent to the verification server. Instead of cryptographically secure, the hash must be very efficient to allow for greater code coverage with minimal system impact. Thus, DARTS incorporates the same methodology for hashing that is used by Seshadri et al. in ICE – that is, an alternating series of ADDs and XORs.

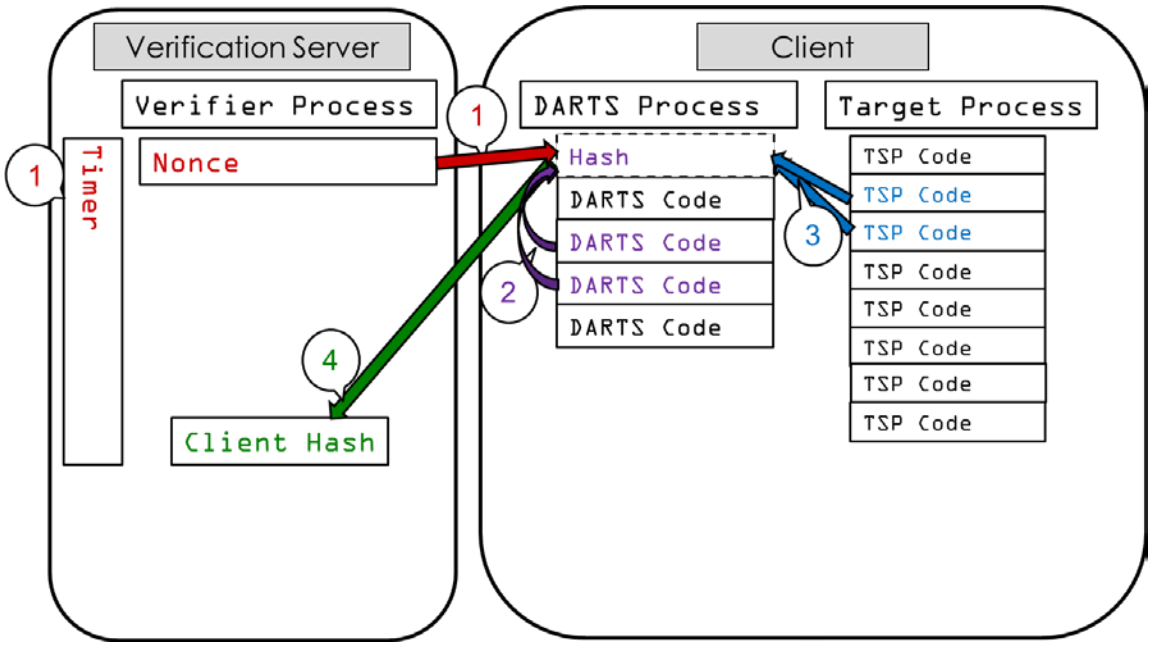

Figure 1: Overview of DARTS process. 1) Server generates and transmits nonce; starts timer. 2) DARTS uses the nonce to select memory locations and hashes its own code. 3) DARTS uses current hash to select and include TSP memory locations in the hash. 4) DARTS transmits completed hash back to server

When attesting the entire program and associated Dynamically Linked Libraries (DLLs) - in this case 2.1 MB - DARTS includes all memory addresses sequentially, using

the nonce as the initial value. In other cases, where DARTS is only attesting a portion of the program's memory, DARTS used a pseudo-random number generator seeded with the nonce value to determine the addresses to include in the hash. This random approach makes it difficult for any code to hide itself. Although not strictly guaranteed, but with a high degree of certainty, all memory locations will eventually be inspected. Alternatively, DARTS was configured to use a hybrid approach wherein the nonce is used as a random seed which determines a starting memory address to begin each iteration with all subsequent memory reads being linear. Speculation is that this approach may improve caching performance by taking advantage of spatial locality.

There are some previously demonstrated attestation techniques that have been consciously omitted from the design of DARTS. For example, ICE integrates the address of the referenced memory in the hash in order to protect against attacks. However, including this feature currently makes attestation infeasible in a system with dynamic memory allocation. The result of this feature's omission is that the DARTS algorithm can be defeated by sophisticated malware on systems with adequate excess memory space. However, DARTS significantly increases the cost and complexity for the attacker, as well as the probability of detection.

3.3 Assumptions and Limitations

First, it is assumed that all layers of the system stack have not been tampered with. This includes the operating system, kernel, firmware, and hardware. As any layer in the system stack is entirely dependent upon and can be fooled by the layers beneath it, such modifications defeat application-layer attestation. The scope of software attestation

can be expanded to encompass the operating system (OS) and kernel, which would enable this assumption to no longer apply to OS and kernel code, but this is beyond the scope of this thesis. Additionally, other attestation schemes can be performed before run-time to verify that the system started in a tamper-free state and since hardware and firmware modifications typically cannot be performed without impact to the system, the likelihood of hardware or firmware being altered while the system is in operation is minimal.

Second, the DARTS code is assumed to be fully optimized. Software-based attestation relies upon fully optimized code in order to ensure that malicious code cannot use a faster implementation to compute a response with enough extra computer cycles to hide or move code. Due to the fact that DARTS is a representative expansion of software attestation, going through the effort to demonstrably optimize the code is of minimal academic or practical value.

The third assumption is a tenet of software attestation and is closely related to the second assumption: there is no other, more powerful, untrusted computer on the network. This assumption is necessary because if both machines were compromised, it would be possible for the target to act as a relay and offload the attestation process to the more powerful system. The more powerful system would then complete the attestation more rapidly than the target and could therefore allow the compromised system to return the correct reply within the allotted time limit.

The fourth and final assumption is that the system's highest priority is available to DARTS. In order for attestation to work, the method must be atomic – that is non-interruptible. Any context switch, where the attesting method loses control over the

processor and its memory is an opportunity for malware to hijack the method and produce false results.

3.4 Evaluation of Success

In order to evaluate whether DARTS has achieved its objectives, it is necessary to state what criteria will be used to distinguish between success and failure. To determine that attestation can be performed at run-time on a real-time system, the target system process' start time and execution time must remain within one standard deviation of values collected without ongoing attestation. The determination of whether or not software attestation is possible on systems using a Von Neumann architecture and dynamic memory allocation will be based on whether or not DARTS is able to perform attestation, regardless of impact, on systems that use dynamic memory allocation and Von Neumann architecture.

3.5 System Configuration and Setup

This proof-of-concept is implemented in two single-core systems-in-a-box systems running Ubuntu 14.04 and the Real-Time Linux patch, version 4.4.12-rt19. This system configuration allows for nanosecond-level real-time scheduling based on priority. In order to utilize timing in a distributed setup, it was also necessary to establish a Network Time Protocol (NTP) server on one of the systems, with the second system operating as an NTP client.

The function of the program being attested is not of particular concern; it is representative of a real-time process that cannot be allowed to sustain disruption. A simple real-time program called Target System Process (TSP) was created that shuffles

values among four different variables and stores start and stop timestamps in an array (any activity that could not be optimized away by the compiler would suffice). In order to facilitate this program's ability to simulate a critical real-time process, its priority was set one below the maximum level within Real-Time Linux, leaving the highest priority for DARTS to ensure atomicity.

3.5.1 Memory Parsing

Software attestation is generally best applied in a Harvard architecture, in which data is separated from executable code, and on systems with static memory allocation. Because Linux systems are built upon a Von Neumann architecture with dynamic memory allocation, additional steps were necessary in order to accommodate both the presence of data in the attestation space and dynamically addressed execution code. A key requirement was to identify which addresses allocated to TSP contained data. To satisfy this requirement, it was assumed that all addresses which were volatile (changed during process execution) contained data. Another key requirement was to identify the stack and exclude it from the hashing algorithm because the stack is designed to change during runtime and would make knowing the program's exact state, including all variables, a necessity for attestation. Since most real-time systems gather data that cannot be predicted ahead of time, this type of knowledge is not possible. In order to identify changes from one run to the next, DARTS was modified to print all intermediate values to a file and initiated a series of trials using a single, static nonce against a single, continuous instance of TSP. As a result, volatile values were able to be identified by looking for the deviations from previous trials. In order to ensure consistent performance in spite of the system's dynamic addressing scheme, this address's relative position

within memory (e.g., 990th output) was recorded rather than the address itself and DARTS was programmed to not integrate that line into the hash.

Due to the dynamic addressing scheme employed by most modern operating systems, including the test bed, it was then necessary to end the instance of TSP and start a new instance in order to evaluate consistency of output. As expected, the final hash values were not the same. Removal of volatile values had helped establish consistency within an instance of TSP, but it is speculated that each time the program was initialized, internal references and jumps would be given different offsets to account for the dynamic memory allocation. In order to establish initial functionality, these memory sections were identified in the same manner used to identify the volatile sections and then excluded from the hash. Addresses excluded due to dynamic memory allocation account for 0.3% of the total code. A more permanent and thorough solution will be to have DARTS deterministically compute the changes in the offset and account for it when attesting the program, but research into this area is left to future work. Once completed, this would allow for attestation of a dynamic system without excluding sections of code.

3.5.2 Nonce Generation

Deterministically generated nonce values were used to verify that DARTS produces consistent results by seeding the random number generator on the server with a pre-selected value and performing a series of 1,000 attestations against the client to generate a table of nonces and their associated hash responses. These attestations, or trials, were then re-accomplished using the same seed value, and the second set of results was compared with those from the first sequence. In order to test the program's ability to detect changes, the code was then altered in a variety of ways and the trials were run

again while verifying that the new responses were different than the un-altered code. This confirmed that when run at full capacity, DARTS detects even miniscule changes within either its own code or the target program's code and is not affected by dynamic memory allocation. To reduce the overhead involved with attestation, the portion of DARTS code attested on every round was reduced to 131 KB, which represents 1/16 of the attestation client code. For the remainder of this thesis, when sizes of attestation are mentioned, they refer solely to the amount of TSP code that is being attested since the amount of DARTS code remains constant. The rate at which DARTS is able to successfully identify alterations to its code is the subject of future work.

In scheduler-based real-time systems, the system's ability to operate consistently without interruption can be critical, including loss of life, for example in medical applications or national security. Since DARTS must run on the real-time system in order to determine if the system has been altered, this requires adding a new process into the schedule. To evaluate whether attestation could execute on the target system without interfering with the critical processes, the first step was to assess the program's runtime to determine if it was possible for attestation and TSP to share time on the system's processor. These runtimes are displayed in Figure 2. With 2.1 MB attestation requiring 380 milliseconds, the expectation was that DARTS would be able to run on an operational system without impacting system functionality. Reducing the amount of attestation performed had a notable decrease in execution time for DARTS, but as the attestation was reduced to below 131 kilobytes, the amount of time saved decreases by less than 10% whereas amount of addresses being attested is decreased by 50%, meaning the amount of data attested was halved with minimal time savings. This is a result of the

decrease in the number of addresses attested by DARTS with no change to the overhead required for DARTS to run, increasing the percentage of total runtime that is dedicated to necessary, but non-productive, code execution.

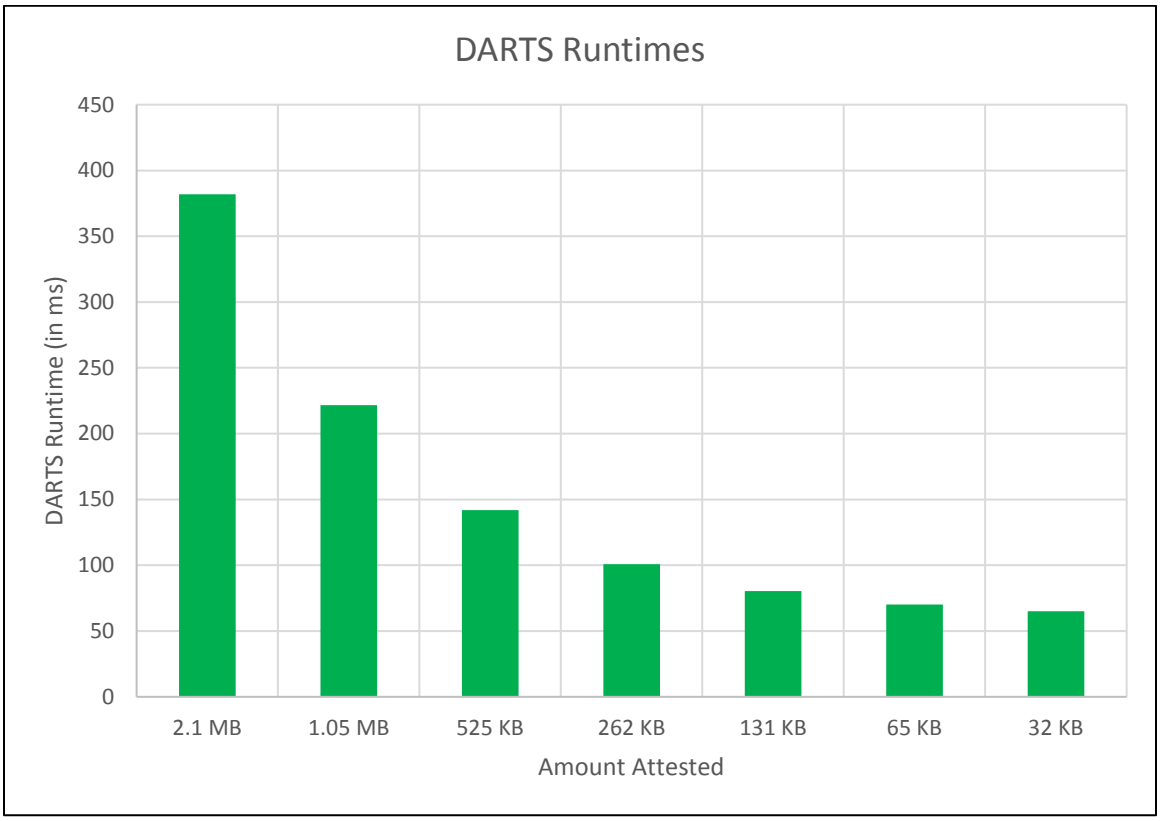

Figure 2: A comparison of DARTS execution times, varying in amount of TSP code attested

3.6 Threats and Attack Methods

Since the intent of software attestation is to verify that a program running on the remote system has not been compromised, it is necessary to evaluate at a high-level the forms of attacks that can be performed against a system and how DARTS in particular responds to these attacks. Though it is still possible to successfully compromise a

DARTS system, it requires more intricate knowledge of the system and the options available to "hack" are greatly reduced.

It should be noted that DARTS does not integrate the stack into the hash, nor does it currently provide any sort of verification of the stack's contents at this time. As a result, an adversary whose malware exists solely on the stack would elude detection by DARTS.

In the case of either simple corruption or unsophisticated malware (i.e., malware that does not try to evade detection), software attestation's function can be simplified to reduce the program's time while maintaining a high assurance that errors will be caught. In such a case, the random nature of attestation is superfluous, and can be removed. This results in a linear (though possibly disjointed) method of attestation that starts at the beginning and continues attestation until either completion or until a set number of addresses have been read (to reduce the amount of time each iteration takes). The next attestation cycle would simply start where the previous cycle ended. This approach benefits from performance bonuses as a result of caching, which provides increased speed. Additionally, it is guaranteed that the entire code space will be attested in n cycles, where $1/n$ is the amount of program data attested.

In the event of an infection from intelligent malware, however, this simple strategy might not be as successful if the malware could determine which sections of code would be attested next because it could simply move to a different section that would not be attested. Because of this, random address selection is essential. Using the nonce value as a random seed, the attestation program selects addresses at random to attest. Since the sequence is determined by the nonce, the transmission of which has

already started a timer, the malware would not have adequate time to hide. The use of random addresses is able to defeat intelligent malware, but results in decreased code coverage performance. Without careful deconfliction of target addresses between attestation cycles, addresses could be read inefficiently, with some being read several times before others are attested at all. This produces gaps in coverage that could enable lucky malware or corrupt data to remain undetected longer than is desired. The best case scenario for full program space attestation remains at *n* cycles, but as the address space grows, the probability of reaching this best case diminishes in a manner consistent with the birthday problem [23]. Additionally, this type of approach does not benefit from caching because each subsequent address has no deterministic relation to its predecessor.

One way to alleviate the caching issue is to use a random generator to select a new starting point every cycle, however subsequent addresses would still be read linearly. This allows the attestation scheme to benefit from caching and also helps alleviate the uneven read of addresses while retaining the unpredictability of using a random start address. Due to the atomic nature of the attestation process, the linear nature of the address reads is of no concern because any malicious behavior to evade would require interruption of the attestation in order to execute, which would introduce a detectable change in timing, and thus be ineffective.

One method that can be employed by malware to successfully evade DARTS in its current form would be to clone both DARTS and the target program in memory. The malware would then run the attestation normally against the preserved clones while the malware manipulates and alters the actual running copies of the programs. This requires the system to have enough memory for two copies of each program. ICE, by Seshadri, et

al., addresses this by including both the program counter and memory addresses loaded directly from the CPU registers, which is a feature that could be built into DARTS assuming the issue of dynamic memory allocation can be addressed (see future work).

3.7 Summary

As run in the experiments, DARTS is an attestation protocol designed to attest a program running on a real-time Linux system with the intent of evaluating the feasibility of running attestation on a) a real-time system and b) a system with dynamic memory allocation of a Von Neumann architecture. In short, DARTS is designed to increase an adversary's cost and complexity when attacking a specific system.

4. Analysis and Results

4.1 Chapter Overview

This chapter discusses the results of the experiments as well as their implications.

4.2 Results of Simulation Timing

The first step to determine whether or not attestation has any impact on the operation of the system was to evaluate TSP's performance without attestation. TSP was scheduled to begin exactly at 0 ms, and Figure 3 displays the actual start times for TSP. The average delay between scheduled and actual start time was 12.003 µs with a standard deviation of 1.5 µs. Cause of this delay is suspected to be context switching (i.e., loading of the program into cache). Total execution time for TSP remained consistent at 1.2 µs.

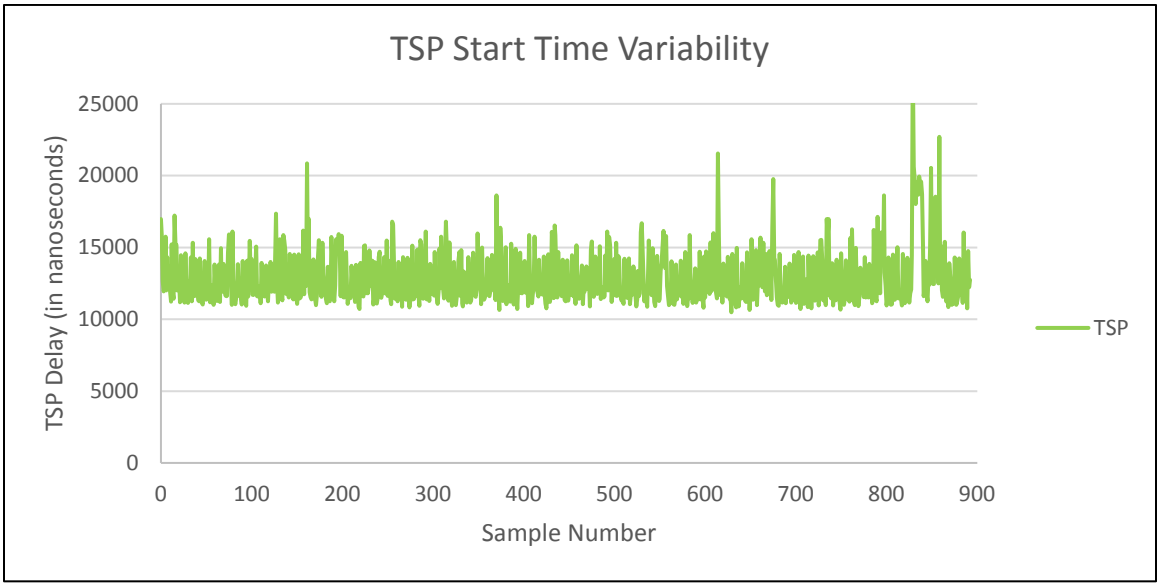

Figure 3: TSP start time delay while unattested. Standard deviation: 1.5 µs; average start time delay: 12 µs.

In order to find the optimum time to execute attestation, TSP was set to activate when the system clock read 500 milliseconds in order to more easily visualize the effects

that attestation had on the program's runtime. The start time of DARTS was then incremented by 1 millisecond per trial starting at 0 milliseconds, forming something of a convolution between start times and TSP delays, and displayed in Figure 4. As expected, this exercise revealed the existence of certain time periods where attestation had a negative impact on system operations – specifically when the offset for DARTS was between 128 ms and 500 ms. As the start time for DARTS was delayed incrementally during this period, the observed disruption became increasingly severe until it peaked at the full runtime for DARTS. This was a result of the decreasing amount of time between the start time of DARTS and the scheduled start time for TSP. Since the system utilized a single-core processor and the scheduler, as configured for this experiment, will not interrupt attestation for any reason, TSP had to wait for DARTS to complete before it was allowed to execute. In real-time systems, these situations must be explicitly avoided. During these trials, attention was paid to the runtime of TSP as well as start time delay in order to make sure that attestation did not cause the program's runtime to increase unexpectedly.

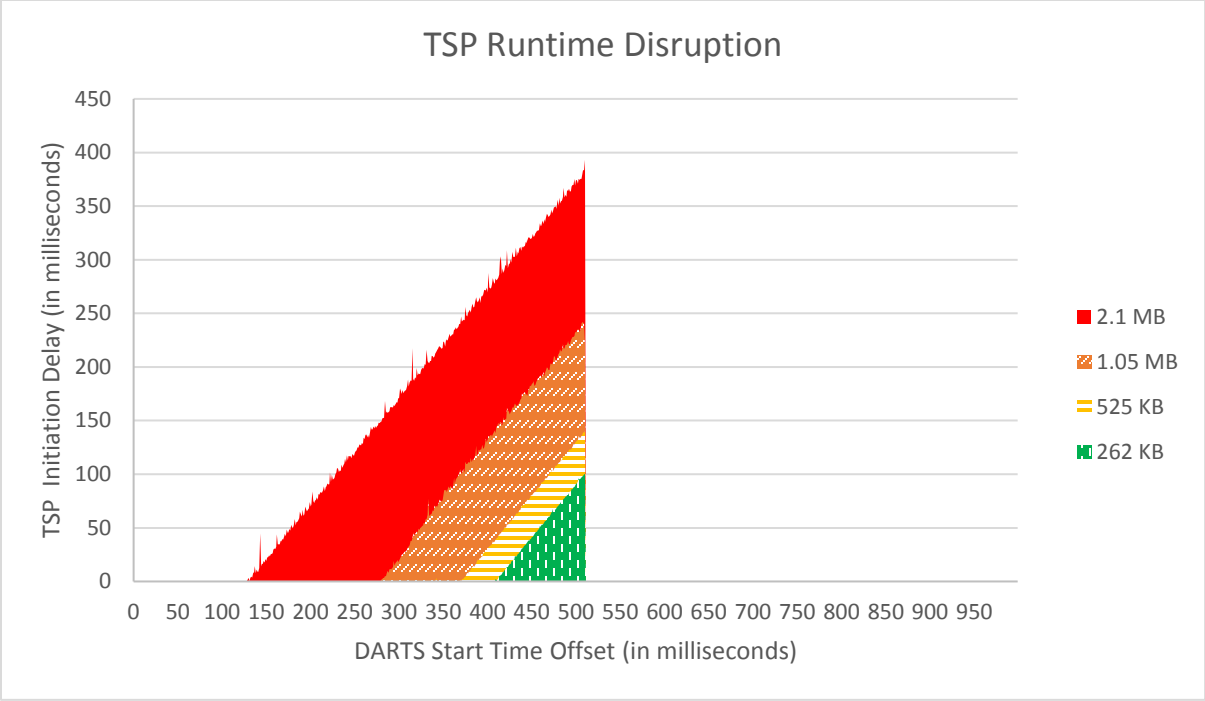

Figure 4: Convolution of TSP's runtime disruption. As DARTS' start times approach 500 ms, the amount of delay suffered by TSP increased. Attestation requests after this period suffered no visible delay due to TSP's task being accomplished prior to attestation.

4.2.1 Process Deconfliction

At first glance, Figure 4 suggests that even attestation of 2.1 MB can be run on systems without impacting critical processes. When focusing in on one such region where DARTS appears to have no impact, such as when the start time offset was between 520 and 770 ms, it becomes evident that there was some delay incurred by TSP. As Figure 5 shows, TSP start times during this period remained relatively consistent around the 15 µs mark, but this was still 2 standard deviations (3 µs) higher than the average of 12 µs for non-attested TSP, which does not meet the requirements to be non-disruptive. Considering this was the period of least disruption caused by the 2.1 MB attestation, this reveals that in order for attestation of 2.1 MB to be feasible during any period,

modifications are required. All other attestation sizes resulted in delays nearly indistinguishable from the 2.1 MB attestation, as can be seen in Figure 6. The fact that all four attestation sizes, regardless of total runtime, had the same impact on TSP's timing during this period suggests that the interference was an indirect result rather than direct impact (i.e. partial flushing of TSP instructions/data from the cache as opposed to monopolizing CPU cycles).

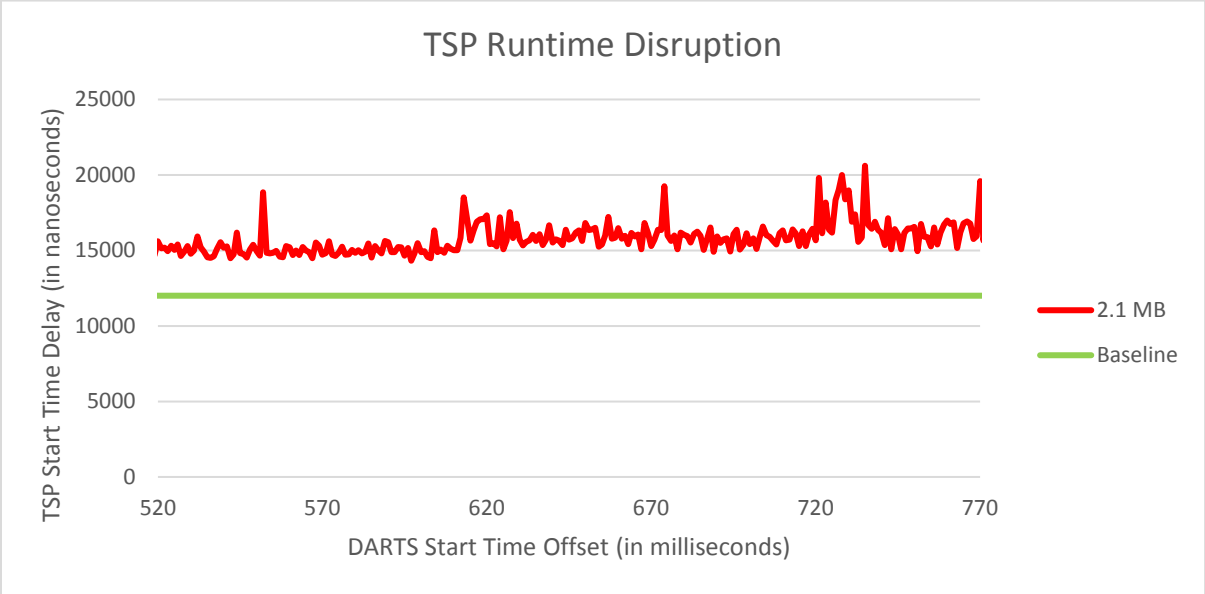

Figure 5: TSP start time as affected by 2.1 MB attestation when attestation was using a 520 ms to 770 ms offset. Baseline represents the average delay for un-attested TSP.

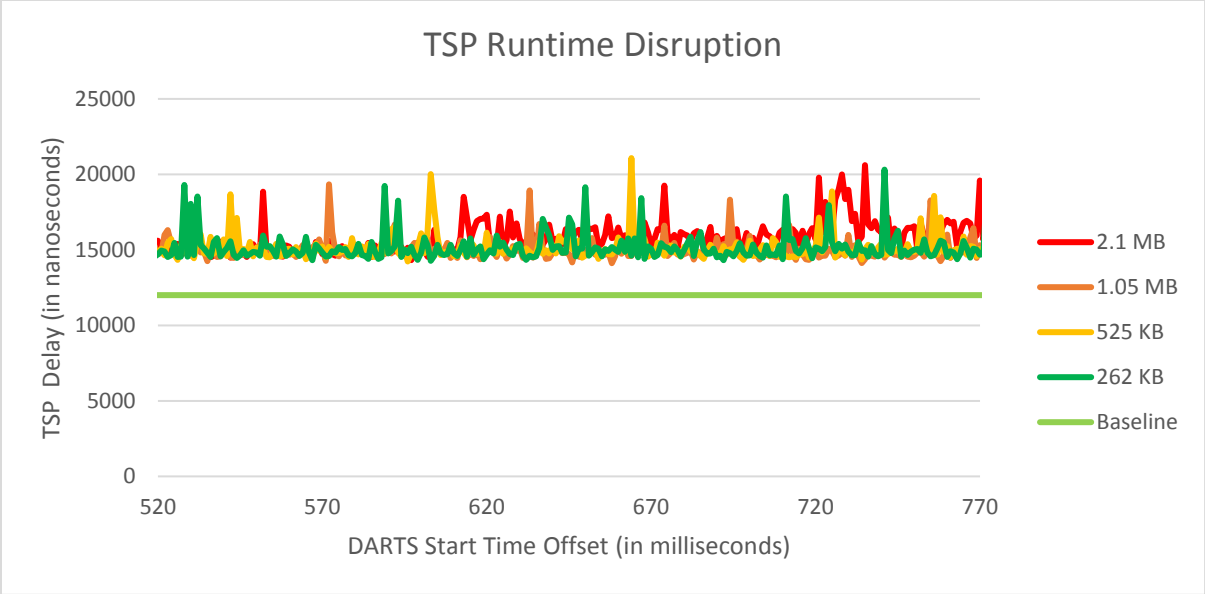

Figure 6: TSP start time disruption – comparison of TSP's delay as affected by 2.1 MB, 1.05 MB, 525 KB, and 262 KB attestations with the average delay for TSP.

Considering that the peak run time of DARTS was equal to DARTS' total runtime and occurred when DARTS had a 500 ms execution offset, all later offsets for DARTS resulted in the attestation occurring *after* TSP completed its time-critical task. Coupled with the fact that all attestation sizes incurred the same delay during this time period, this suggests that the delay suffered by TSP (roughly 3 µs) was primarily the result of TSP's program code being replaced in cache, thus incurring cache miss penalties. When attesting a real-time program running alone on a system, it was not expected to have to account for cache miss penalties incurred by the intrusive flushing of the cache that occurred as a result of DARTS. On a system where the interval between trials (in this case, 1 second) is all that matters, the cache miss delays would not have a negative impact because all iterations would suffer comparable delay and result in the interval not being affected.

4.2.2 Adjusted TSP Start Time

In other systems where the execution time itself is essential, such as when the system is part of a networked infrastructure requiring precise timing, this delay becomes unacceptable. This fostered the idea that the scheduled runtime for TSP could be altered in anticipation of the cache miss delay. If it is necessary for a system to begin its execution at exactly 500 ms, and as a result of attestation requires 5 µs longer to initialize, then scheduling the program to run 5 µs earlier should at least reduce this delay. Figure 7 shows TSP's average, unattested start time compared with the delay incurred by 2.1 MB attestation if TSP's start time were rescheduled 5 µs earlier to account for the newly introduced cache miss penalty. This resulted in a reduction of the delay to within 1 standard deviation of TSP's average runtime.

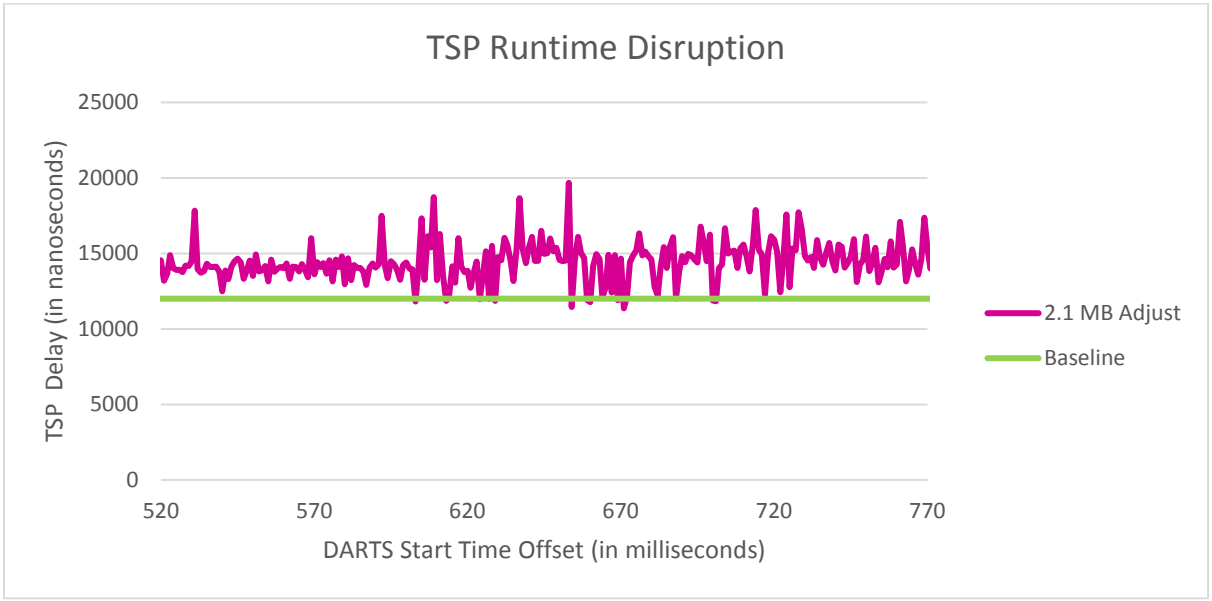

Figure 7: TSP timing disruption comparison – standard TSP delay compared with 2.1 MB attestation trials against an adjusted TSP start time.

The intent was to minimize or eliminate the impact on TSP's execution time, so the offset with the lowest interference represented from the previous trials (540 ms) was selected. DARTS was then configured to use this new offset for another 1,000 trials in conjunction with TSP's 5 µs early start time. The results were then compared to the TSP's start time without attestation and displayed in Figure 8. Not only did the delay incurred by TSP remain within 1 standard deviation, but was far more consistent throughout the trials than TSP when it was not being attested. With a standard deviation of 0.5 µs, the new standard deviation was 1/3 that of TSP when running alone, meaning that not only would attestation ensure proper operation of the system, but there are some cases where the system's operational stability would actually benefit.

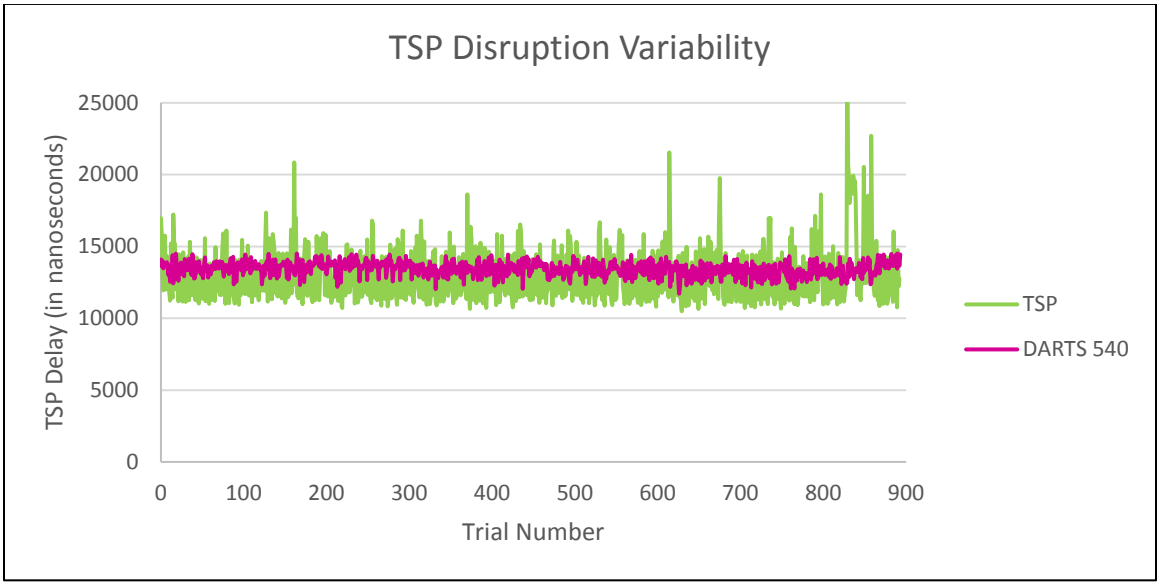

Figure 8: Visualization of the delay caused by DARTS running at an offset of 540 ms with TSP scheduled 5 µs early compared to unattested TSP.

This trend of reduced standard deviation in start time delay was noticed uniformly across all trial series and attestation sizes and even during periods where TSP suffered

direct interference from DARTS. When TSP runs on the system without attestation or other programs, the cache hit rate appears to be non-uniform, which resulted in a fluctuation in start times. When attestation is introduced, however, it is almost guaranteed that TSP has to completely reload its code into the cache resulting in a more consistent initialization sequence, which in turn results in lower start time variation, albeit with greater total execution time.

4.3 Attestation Coverage and Effectiveness

In order to fully assess the usability of attestation, it is also necessary to evaluate how effective attestation, in particular DARTS, is when the volume of attested data is reduced to accommodate for shorter execution periods. The success rate of reduced-volume attestations depends greatly on the method used to determine which addresses to inspect, which was briefly discussed earlier in Section 3.5. A representative sample of 1,024 address spaces was selected for this evaluation in order to keep the data processing at a manageable size. Figure 9 displays a comparison of the coverage provided by the three primary methods. This graph is consistent with statistical analysis for both linear reads (where no statistical analysis is required) and true random, where the birthday problem's collision equation[1], $E = \sum_{k=1}^{n} n - t + t\left(\frac{t-1}{t}\right)^{n}$, accurately predicts the number of collisions that are expected. The values resulting from Equation 1 are compared with the experimentally determined values and displayed in Table 1. Each collision results in a duplicate read of an address, reducing the total number of unique

[1] E is the expected number of collisions, *n* is the number of addresses being read on a given pass, and *t* is the total number of available addresses.

addresses incorporated and therefore reducing the total amount of memory attested on a given trial.

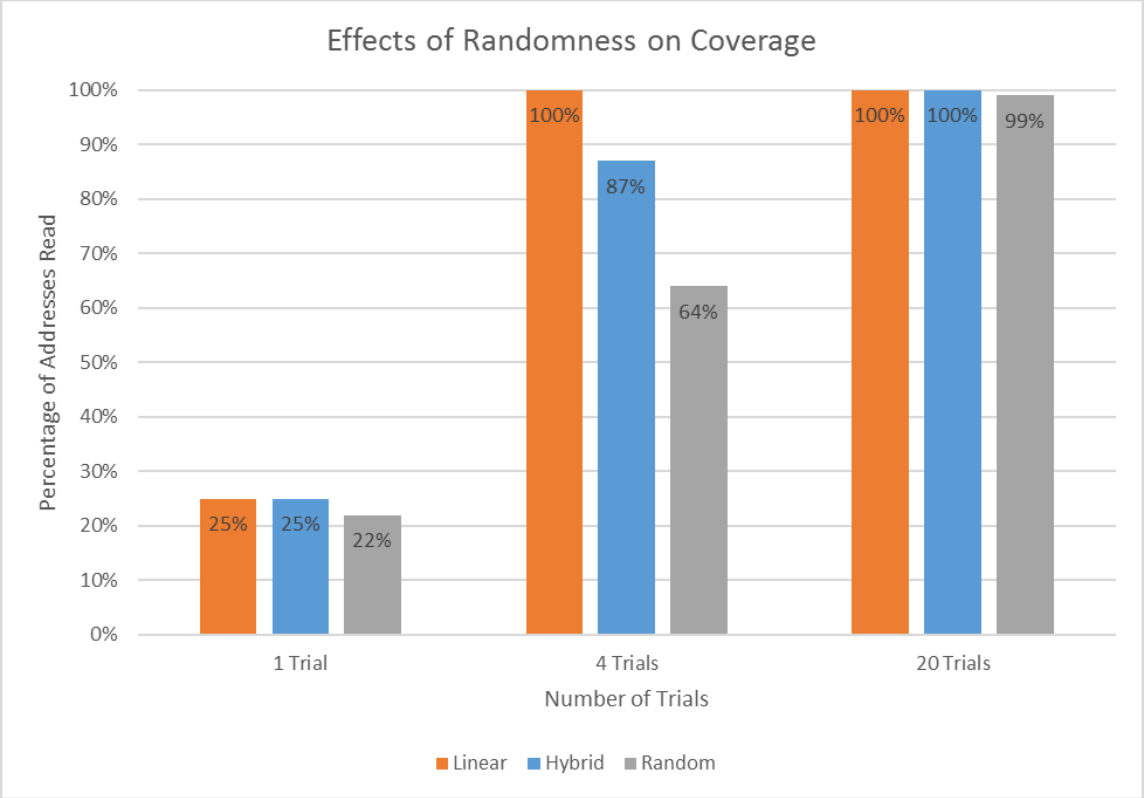

Figure 9: A comparison of the coverage provided by different methods for integrating randomness into attestation.

Table 1: Comparison between predicted and actual values for random address selection.

Number of Trials	*Statistical Prediction*	*Experimentally Determined*
1	227 unique / 29 duplicate	228 unique / 28 duplicate
4	648 unique / 376 duplicate	652 unique / 372 duplicate
20	1018 unique / 4102 duplicate	1015 unique / 4105 duplicate

The first method that could be utilized if intelligent malware is not a concern is also the most effective – a simple linear approach to attesting the address space. The

number of trials required to guarantee that any given address will be attested is equal to *n*, where 1/*n* is the fraction of addresses being attested. This method also benefits from cache pre-loading to further reduce the speed required to perform attestation.

The counter to linear attestation is a completely random scheme. Using a random approach, DARTS selects each successive address randomly, with possible repetition. Though it comes at a cost, random address selection makes it possible to detect intelligent malware that might be able to sense an impending attestation and relocate itself. As each memory address is independently read from one another, caching becomes more of a liability than a benefit. Additionally, due its randomness, the average number of trials required for a given location to be read is the largest out of the three methods. Using the equation to determine the probability an event will occur[2], $P = 1 - \left(\frac{t-1}{t}\right)^n$, determined that using a random setting results in only a 22.1% chance for each address being read, rather than the expected 25% when ¼ of the address space is attested. Using this probability (22.1%) in the equation for infinite series[3], $C = \frac{P}{1-P} * \sum_{n=1}^{20} n\,(1-P)^n$, revealed that the average number of trials required for all addresses to be attested was 4.35 over the course of 20 trials (increasing to 4.52 for an infinite series), compared to 4 for straight linear. Figure 10 displays the actual memory reads from a representative sample using this full random method.

[2] *P* is the probability that an address will be read, *t* is the total number of addresses available, *n* is the number of addresses read per cycle

[3] *C* is the expected number of cycles for a given address to be read, *P* is the probability that any given address will be read, *n* is the number of rounds of attestation / iterations through the equation

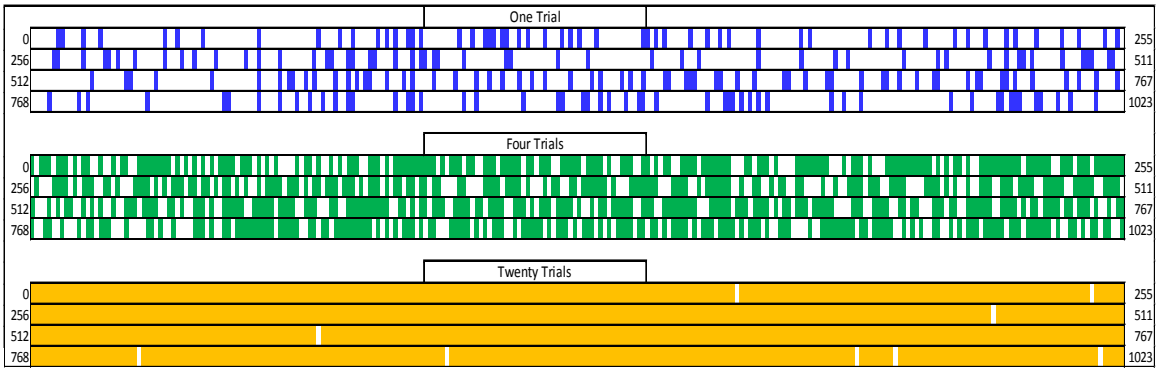

Figure 10: Memory locations read after 1, 4, and 20 trials using a full-random discovery scheme.

Alternatively, a hybrid approach could be used that would combine the benefits of linear and random memory reads. In this hybrid approach, DARTS would use the nonce to generate a random address to read. From this point on, the address reads would be linear. This allows DARTS to benefit from the caching pre-fetch and eliminates the need to generate a new random number for each address to be read while still allowing the program to capture intelligent malware via unpredictability. The unpredictability comes from the nonce that is used to generate the random starting location and the fact that a timer has already begun on the remote server before the nonce even arrives at the client machine to be verified, leaving malware no time to relocate itself to a safe location. While the random nature does result in uncertainty, the equation for an infinite series maintains that the average number of cycles required to read a given address is 4. Figure 11 displays the actual memory reads from a representative sample using this hybrid method. It should be noted that the same hit rate could be achieved with true random if the random generation did not allow for duplicate addresses. However, doing this would

increase the time required for the process to run without producing any meaningful benefit.

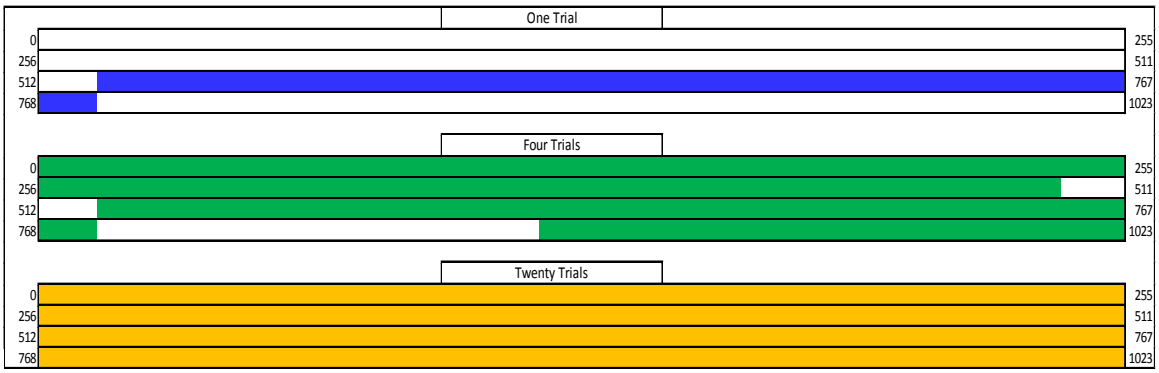

Figure 11: Memory locations read after 1, 4, and 20 trials using a hybrid approach (random start location selection followed by linear reads).

4.4 Performing Attestation on a Dynamically Addressed System

As was mentioned in Chapter 3, it was necessary to identify and exclude certain memory offsets in order to enable consistency between all instances of the TSP. While this enables DARTS to attest and return consistent results, it also leaves certain addresses unattested, and as such would provide an easy opportunity for malware to evade detection. To alleviate this vulnerability, analysis has been performed on the changes that occur between iterations of the target program in order to identify patterns usable by DARTS to account for changes resulting from dynamic addressing.

Using DARTS' ability to locate the sections of code as they are loaded in memory, four contiguous blocks of data were identified. DARTS was then configured to do a full linear read of the memory contents and print the un-altered values to a file along with the starting value of the block and offset for each 4 byte memory location. For the

sake of this analysis, only one block of TSP's code was analyzed during this stage. This was done to keep the data set at a size conducive to the initial evaluation.

Analysis of the values from this block revealed that the contents of only 15 out of 1,024 memory locations change because of the dynamic addressing. Further evaluation of the values within these 15 address spaces has revealed that the 2nd through 11th altered locations retain a constant relation to the first (for example: the 2nd location is always equal to the first, the third is always the first plus 1,024, etc.). Such consistency supports the supposition that the changes in values is due to variations in jump offsets as a result of dynamic memory allocation not maintaining relative locality for the code between iterations.

Using these results, it is possible to program DARTS to account for some of the dynamic shift between iterations. The result of this is that certain addresses do not have to be skipped simply because they change between instances of TSP execution – DARTS is able to capture the value from the first memory location and subtract it from the values within locations 2 through 11, incorporating the remainder into the hash. This provides greater assurance that the values have not been altered.

The difficulty for malware to circumvent this will be greatly increased if the value of the 1st location is able to be determined prior to the memory read because DARTS' ability to account for the change in values would not depend on any data values that might have been altered by the malware. Analysis of the binary values for the 1st location reveal that the first 12 bits as well as the last 12 bits are consistent across all iterations of the program. Because of this, the middle byte holds the key to enabling DARTS to

identify the expected value independent of the actual value. As of yet, no discernable pattern has been identified with regards to the number's selection.

4.5 Investigative Questions Answered

With regards to the investigative questions, DARTS demonstrated the ability to successfully perform software-only attestation on a real-time system. In the event that the amount of attested memory has to be altered to accommodate system requirements, the number of trials required to return to a given memory location is, on average, inversely proportional to the fraction of memory space being attested (i.e., if ¼ of the memory was attested, then it would take an average of 4 trials to attest every address). This was statistically determined and experimentally verified. This does not hold for methods that allows duplicate reads of the same address within the same attestation cycle. The performed experiments also suggest that programming software attestation to accommodate for dynamic memory allocation is possible, though further work is required to enable DARTS to anticipate all values based on the memory addresses.

4.6 Summary

Experiments have demonstrated that real-time attestation is possible as long as there is a sufficiently large period of time on the system where the processor is free from critical processes. If caching is utilized on the system, the system owner might need to account for newly introduce cache miss penalties to ensure timing remains consistent for networked services, but the interval will remain consistent.

5. Conclusions and Recommendations

5.1 Chapter Overview

This chapter discusses the conclusions and implications of the research as well as recommendations for future works.

5.2 Conclusions of Research

Through this initial proof-of-concept series of experiments, it was shown that real-time software attestation is possible, at least in some situations. As suspected when starting these experiments, deconflicting the run times for the critical process and the attestation program is necessary, but there is more to run-time attestation than this. A key side effect of running attestation on a system with limited tasks is that the cache, which would otherwise maintain at least some data for the critical process, results in more cache miss penalties. In systems where the scheduling of the critical process can be altered, the system designers can schedule the critical system process to execute earlier to offset the inevitable delay, thus maintaining system integrity.

An interesting byproduct of attestation was that the standard deviation of the critical process' start time delay was reduced by 2/3. Though this might appear counter-intuitive, having the cache flushed between each iteration greatly reduces the inconsistency presented by general computer functions' impact on the cache. The end result of this is that the addition of attestation both ensures proper operation of the system and enhances the consistency of the critical process. It is understood that this might not always be the case.

5.3 Significance of Research

This proof of concept demonstrated that systems can be attested without bringing them offline. Whereas it has previously been determined that attestation through software-only means is possible, previous attestation methods require interruption of the system's operation. Critical infrastructure systems, such as power, gas, and water treatment, as well as sensors aboard aircraft, to name a few systems, do not handle interruptions well, providing limited opportunity for attestation. DARTS enables ongoing attestation of such systems, further reducing the impact of data corruption or adversary intrusions, provided that the system has resources capable of accommodating attestation.

Additionally, altering the amount of data that is attested during each iteration of the attestation program allows for greater flexibility and customizability to fill a greater variety of roles. The tradeoff between faster runtimes and more complete attestation creates a sliding scale that can be used to tune attestation to the needs of the system. If the urgency of malware detection is low, then attestation could be scaled back to reduce resource costs while allowing for probable detection of malware within a time limit determined by the system's owner. On the opposite end, if malware must be detected nearly instantly, the amount of attested data can be increased to the maximum amount while remaining non-interferent with the system.

5.4 Recommendations for Future Research

At this time, the research into performing software attestation on a dynamically addressed system remains incomplete. Although this thesis was able to deterministically

predict and therefore account for some memory contents altered by dynamic allocation, not all memory spaces were able to be evaluated. Further analysis is needed in order to completely enable DARTS to account for the dynamically altered values. Additionally, as DARTS does not incorporate the memory location's address into the hash as ICE [8] does, DARTS has no defense against intelligent malware that creates an un-altered copy to use for attestation. Adapting a similar solution to DARTS, would greatly increase an adversary's cost to defeat software attestation on dynamically addressed systems.

Due to the exclusion of volatile memory addresses from the hash, the algorithm remains imperfect. This is partially a result of not knowing the exact values to expect in a real-time system. Another aspect of future work that should be investigated is to evaluate the volatile address spaces (the stack in particular). By evaluating the contents of the stack for style rather than exact value (e.g., if an integer is expected in a particular location relative to the bottom of the stack, is that what is found?). This would enable the program to identify malware that resides on the stack.

Another experimental alteration that could be incorporated is the use of dual and triple nonces. Using separately computed values would vastly increase the difficulty in pre-computing hash values. The first would be an initialization for the hash itself, while the second would be used as the random seed in determining sequence of memory reads, and the third value would be used to alter the sequence of ADD and XOR used to generate the hash. Keeping each nonce a 4-byte integer, this would increase the required size of a pre-computed table from 32 Gigabytes to 1,048,576 Yottabytes (1×10^{18} Terabytes), if using the minimum 16 bytes per entry with no formatting; in other words, impossible with current technology.

Current software attestation methods rely upon the atomicity of attestation to ensure that malware was not able take control of the processor and produce false results. However, should a conflict arise between ongoing attestation and the critical process that is being attested, it is likely necessary that attestation must be pre-empted by the critical process to avoid system failure. This would result in a correct response, but invalid response time from the client and therefore the client would be deemed compromised. Investigation into the feasibility of allowing occasional delayed responses from the client could not only be used to guarantee the stability of a real-time system during attestation, but also would allow for interrupt-based real-time systems to benefit as well.

5.5 Summary

Dynamic attestation of real-time systems is possible, and alterations to the amount of data attested allow attestation to be more closely tailored to the needs of the system. Attestation on systems with dynamic memory allocation looks promising, as values can be deterministically anticipated and accounted for, but requires further research before it reaches operational capability.

Appendix

Appendix A: TSP Code

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <sched.h> //RT schedular
#include <sys/mman.h>
#include <string.h>
#include <sys/resource.h> //allows editing of rlimits

#define MY_PRIORITY (98) /* 99 is the maximum priority */
#define MAX_SAFE_STACK (4*1024) /* The maximum stack size which is
        guaranteed safe to access without faulting */

void stack_prefault(void){
    unsigned char dummy[MAX_SAFE_STACK];
    memset(dummy, 0 , MAX_SAFE_STACK);
    return;
}

int main(int argc, char* argv[]){
    setbuf(stdout, NULL);
    int a = 2, b = 3, c = 4, trials, initial, interval, temp;
    char outFile[] = "RTwOut.csv";  //output file
    struct timespec t;  //Timing struct that allows us to keep track of secs and nsecs at the same time.
    struct timespec u;  //timing structure to measure consistency
    struct sched_param param;   //RT schedular struct

    //loads variables from the variables file
    FILE *stuff = fopen("variables.txt", "r");
    char foo[256];
    while (fgets(foo, 256, stuff) != NULL){
        sscanf(foo, "a - %i\n", &a);
        sscanf(foo, "b - %i\n", &b);
        sscanf(foo, "c - %i\n", &c);
        sscanf(foo, "trials - %i\n", &trials);
        sscanf(foo, "initial - %i\n", &initial);
        sscanf(foo, "interval - %i\n", &interval);
    }
    fclose(stuff);

    //creates arrays to store data for use in the output
    int randos[trials];
    int start[trials];
    int stop[trials];

    //creates output file
    FILE *d = fopen(outFile, "w");  //clears existing file
    fwrite("", 1, 0, d);
    fclose(d);
    FILE *e = fopen(outFile, "a");  //opens to append
```

```
    param.sched_priority = MY_PRIORITY;
    //sched_setscheduler(0, SCHED_FIFO, &param);
    if(sched_setscheduler(0, SCHED_FIFO, &param) == -1){
        perror("Set schedular failed!");
        exit(-1);
    }

    //Lock out stack memory
    if(mlockall(MCL_CURRENT|MCL_FUTURE) == -1){
        perror("mlockall failed");
        exit(-2);
    }

    //Set up stack
    stack_prefault();

    //Set up struct to wait 2 sec before starting.
    clock_gettime(CLOCK_MONOTONIC, &t);
    t.tv_sec+=2;

    int count = 0;
    int i = 0;
    t.tv_nsec = initial;
    while(i < trials){
        clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &t, NULL);
        clock_gettime(CLOCK_MONOTONIC, &u);
        start[i] = u.tv_nsec;
        t.tv_sec += interval;

        temp = a;
        a = b;
        b = c;
        c = temp;

        clock_gettime(CLOCK_MONOTONIC, &u);
        stop[i] = u.tv_nsec;
        i++;
    }
    //prints array contents to file for use in data analysis
    i = 0;
    while ( i < trials){
        fprintf(e, "%i, %i, %i\n", i, start[i], stop[i]);
        i++;
    }
    fclose(e);
}
```

Appendix B: DARTS Code Excerpts

```
//takes in a memory file, start address, stop address, socket (if there is any), file name (to write data to), and a seed value
unsigned long dump_memory_region(FILE* pMemFile, unsigned long start_address, long length, int serverSocket, char* name, int nonce) {
    unsigned long address;  //current address we're at
    int pageLength = 4;    //32-bit integer
    int current = nonce;    //seed value, only the nonce on round 1
    int temp;       //placeholder for the read data
    FILE *f = fopen(name, "a"); //file to deposit dump info into
    unsigned long end_address = start_address + length;
    int spot;
    int reads = length / (4 * fraction);    //number of reads that will occur
    int i = 0;
    srand(nonce);
    spot = rand() % (length / 4);
    spot = spot * 4;
    int count = 0;      //used to flip-flop + and XOR
    int spotcount = 0;  //used to track movement through spots array
    int firstSpot;
    while (spots[spotcount] < spot && spots[spotcount]!=1){
        spotcount++;
    }
    if (spots[spotcount] == 1){
        spotcount = 0;
    }
    firstSpot = spotcount;
    fseeko(pMemFile, start_address + spot, SEEK_SET);
    for (i = 0; i < reads; i++) {
            fread(&temp, 1, pageLength, pMemFile);
        if (count % 2 == 0) {
            current = current + temp;
            fprintf(f, "%lu, %i, %i\n", start_address, temp, spot);
            count++;
        } else {
            current = current ^ temp;
            fprintf(f, "%lu, %i, %i\n", start_address, temp, spot);
            count++;
        }
        spot += pageLength; //increments the memory address
        if (spot >= length){
            spot = 0;
            fseeko(pMemFile, start_address, SEEK_SET);
        }
    }
    fclose(f);
    return current;
}
```

```
//this part of the code attests DARTS
//if the variable DARTS is set to 1 in the variables.txt
//================================================================================
    if(DARTS == 1){
        FILE *fp = popen("/bin/ps -A", "r");
        while(fgets(process, sizeof process, fp)){
           if(strstr(process, "DARTS")){
            sscanf(process, "%05d\n", &pid);
           }
        }
        //converts the pid from an int to a string for use later
        sprintf(spid, "%d", pid);
        //reads in the files for the memory map and memory contents
        sprintf(mapsFilename, "/proc/%s/maps", spid);
        FILE* pMapsFile = fopen(mapsFilename, "r");
        sprintf(memFilename, "/proc/%s/mem", spid);
        FILE* pMemFile = fopen(memFilename, "r");
        serverSocket = -1;
        count = 0;
        unsigned long expected = 0;
        while (fgets(line, 256, pMapsFile) != NULL) {
            unsigned long start_address;
            unsigned long end_address;
            if(strstr(line, "stack")) {//found the stack - not including
                count++;
            } else {//program data - first round uses the nonce, subsequent rounds use the previous round's output
                sscanf(line, "%08lx-%08lx\n", &start_address, &end_address);    //extracting the address for use in mem dump
                FILE *tfile = fopen(outFile, "a");
                fprintf(tfile, "========,========,=DARTS==,%i\n", count);
                fclose(tfile);
                if (mode == 1){
                    current = dump_darts_rand(pMemFile, start_address, end_address - start_address, serverSocket, outFile, nonce);
                } else {
                    current = dump_darts_region(pMemFile, start_address, end_address - start_address, serverSocket, outFile, nonce);
                }
                printf("count: %i - %i\n", count, current);
                count++;
            }
        }
        fclose(pMapsFile);
        fclose(pMemFile);
        printf("DARTS - Nonce: %i / Returned: %i \n", nonce, current);
    }
//================================================================================
//end of section
```

Appendix C: Sample Memory Map of TSP

Color-coded to highlight contiguous blocks of memory allocation.

08048000-08049000	r-xp	00000000 08:01 3023319	/home/…/TSP
08049000-0804a000	r--p	00000000 08:01 3023319	/home/…/TSP
0804a000-0804b000	rw-p	00001000 08:01 3023319	/home/…/TSP
08fe3000-09004000	rw-p	00000000 00:00 0	[heap]
b75dc000-b75dd000	rw-p	00000000 00:00 0	
b75dd000-b7785000	r-xp	00000000 08:01 8132908	/lib/i386-linux-gnu/libc-2.19.so
b7785000-b7786000	---p	001a8000 08:01 8132908	/lib/i386-linux-gnu/libc-2.19.so
b7786000-b7788000	r--p	001a8000 08:01 8132908	/lib/i386-linux-gnu/libc-2.19.so
b7788000-b7789000	rw-p	001aa000 08:01 8132908	/lib/i386-linux-gnu/libc-2.19.so
b7789000-b778c000	rw-p	00000000 00:00 0	
b779e000-b77a2000	rw-p	00000000 00:00 0	
b77a2000-b77a4000	r--p	00000000 00:00 0	[vvar]
b77a4000-b77a6000	r-xp	00000000 00:00 0	[vdso]
b77a6000-b77c6000	r-xp	00000000 08:01 8132901	/lib/i386-linux-gnu/ld-2.19.so
b77c6000-b77c7000	r--p	0001f000 08:01 8132901	/lib/i386-linux-gnu/ld-2.19.so
b77c7000-b77c8000	rw-p	00020000 08:01 8132901	/lib/i386-linux-gnu/ld-2.19.so
bfc50000-bfc71000	rw-p	00000000 00:00 0	[stack]

Bibliography

[1] A. . Seshadri, A. . Perrig, M. . Luk, L. . Van Doom, E. . Shi, and P. . Khosla, "Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems," *Oper. Syst. Rev.*, vol. 39, no. 5, pp. 1–16, 2005.

[2] A. Seshadri, A. Perrig, L. Van Doom, and P. Khosla, "SWATT: SoftWare-based ATTestation for embedded devices," *Proc. - IEEE Symp. Secur. Priv.*, vol. 2004, pp. 272–282, 2004.

[3] T. Potthoff and S. Graham, "Dynamic Attestation of Real-Time Systems," to appear in *12th International Conference on Cyber Warfare and Security ICCWS 2017*, 2017.

[4] J. Weiss, "Aurora Generator Test," in *Handbook of SCADA/Control Systems Security, Second Edition*, 2016, pp. 107–114.

[5] S. Checkoway, D. Mccoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno, "Comprehensive Experimental Analyses of Automotive Attack Surfaces," *System*, pp. 6–6, 2011.

[6] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente, "On the difficulty of software-based attestation of embedded devices," *Proc. 16th ACM Conf. Comput. Commun. Secur.*, pp. 400–409, 2009.

[7] A. Perrig and L. Van Doorn, "Refutation of On the difficulty of software-based attestation of embedded devices," *Proc. 16th ACM Conf. Comput. Commun. Secur.*, pp. 400–409, 2009.

[8] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla, "Using FIRE & ICE for Detecting and Recovering Compromised Nodes in Sensor Networks," *Program*, no. December, p. 26, 2004.

[9] A. Seshadri, M. Luk, A. Perrig, L. Van Doorn, and P. Khosla, "SCUBA: Secure Code Update By Attestation in sensor networks," *Proc. 5th ACM Work. Wirel. Secur.*, vol. 2006, pp. 85–94, 2006.

[10] A. Seshadri, M. Luk, and A. Perrig, "SAKE: Software attestation for key establishment in sensor networks," *Ad Hoc Networks*, vol. 9, no. 6, pp. 1059–1067, 2011.

[11] a. Seshadri, A. Perrig, L. Van Doorn, P. Khosla, and C, "Using software-based attestation for verifying embedded systems in cars," vol. 4, 2004.

[12] Y. Li, J. M. Mccune, and A. Perrig, “SBAP Software-Based Attestation for Peripherals.pdf,” pp. 16–29, 2010.

[13] M. Park, “Run-time firmware integrity verification : what if you can ’ t trust your network card ? Embedded devices,” no. September, 2011.

[14] M. Leucker and C. Schallhart, “A brief account of runtime verification,” *J. Log. Algebr. Program.*, vol. 78, no. 5, pp. 293–303, 2009.

[15] H. Wasserman and M. Blum, “Software reliability via run-time result-checking,” *J. ACM*, vol. 44, no. 6, pp. 826–849, 1997.

[16] G. Wurster, P. C. Van Oorschot, and A. Somayaji, “A generic attack on checksumming-based software tamper resistance,” *Proc. - IEEE Symp. Secur. Priv.*, pp. 127–135, 2005.

[17] U. Shankar, M. Chew, and J. D. Tygar, “Side Effects are Not Sufficient to Authenticate Software.” pp. 1–25, 2008.

[18] R. Kennell and L. H. Jamieson, “Establishing the Genuinity of Remote Computer Systems,” *Proceedings of the 12th USENIX Security Symposium*, vol. 52, no. 3. pp. 608–616, 2003.

[19] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik, “A minimalist approach to remote attestation,” *Conf. Des. Autom. Test Eur. - DATE’14*, p. 244, 2014.

[20] L. Davi, A.-R. Sadeghi, and M. Winandy, “Dynamic Integrity Measurement and Attestation: Towards Defense Against Return-Oriented Programming Attacks,” *ACM Work. Scalable Trust. Comput. (STC ’09)*, pp. 49–54, 2009.

[21] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth, “New results for timing-based attestation,” *Proc. - IEEE Symp. Secur. Priv.*, pp. 239–253, 2012.

[22] S. Bratus, N. D’Cunha, E. Sparks, and S. W. Smith, “TOCTOU, traps, and trusted computing,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 4968 LNCS, no. 2006, pp. 14–32, 2008.

[23] J. Naus, “An Extension of the Birthday Problem,” in *The Teacher’s Corner*, 2012, pp. 27–29.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From – To)*
23-03-2017	Master's Thesis	August 2015 – March 2017

TITLE AND SUBTITLE

A Proof-Of-Concept For Software-Only Attestation on Real-Time Systems Using Von Neumann Architecture and Dynamic Memory Allocation

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Potthoff, Travis S., Captain, USAF

5d. PROJECT NUMBER
17G385

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)
Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way, Building 640
WPAFB OH 45433-8865

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-MS-17-M-062

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory
2241 Avionics Circle, Wright-Patterson AFB, OH 45433
Attn: Lt Col Patrick Sweeney
Patrick.Sweeney@us.af.mil 937-938-4252

10. SPONSOR/MONITOR'S ACRONYM(S)

AFRL/RYWA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRUBTION STATEMENT A. APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES
This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

To attest is to affirm to be correct, true, or genuine. Applied to software or executable code, attestation is the ability to affirm that the code actually being executed is the code that is expected, unmodified in any way and may be performed in either hardware or software. Current research into software-based attestation has explored the problem of static attestation, or verifying the software that the system loads at boot-time. For many systems, knowing that the system's initial state is valid is insufficient – verification that the system is still in a good state is needed later and without bringing the system offline or interrupting critical processes. This thesis introduces a proof-of-concept method for performing attestation on real-time systems, named Dynamic Attestation of Run-Time Systems (DARTS). DARTS was designed to be sufficiently customizable in order to enable attestation without interfering with system operations. DARTS also has the ability to perform attestation on systems built upon Von Neumann architecture using dynamic memory allocation. A key contribution of this work is that the entire attestation process is performed wholly in software on a real-time system without impacting the operation of potentially critical processes.

15. SUBJECT TERMS
Software attestation, dynamic attestation, real-time, DARTS, run-time attestation, verification

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Graham, Scott R., AFIT/ENG
U	**U**	**U**	**UU**	60	**19b. TELEPHONE NUMBER** *(Include area code)* (937) 255-6565, ext 4581 (scott.graham@afit.edu)

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18